Dissector: Input Validation for Deep Learning Applications by Crossing-layer Dissection

Huiyan Wang
cocowhy1013@gmail.com
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
Department of Computer Science and Technology, Nanjing University
Nanjing, China

Jingwei Xu*
jingweix@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
Department of Computer Science and Technology, Nanjing University
Nanjing, China

Chang Xu*
changxu@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
Department of Computer Science and Technology, Nanjing University
Nanjing, China

Xiaoxing Ma
xxm@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
Department of Computer Science and Technology, Nanjing University
Nanjing, China

Jian Lu
lj@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
Department of Computer Science and Technology, Nanjing University
Nanjing, China

ABSTRACT

Deep learning (DL) applications are becoming increasingly popular. Their reliabilities largely depend on the performance of DL models integrated in these applications as a central classifying module. Traditional techniques need to retrain the models or rebuild and redeploy the applications for coping with unexpected conditions beyond the models' handling capabilities. In this paper, we take a fault tolerance approach, Dissector, to distinguishing those inputs that represent unexpected conditions (beyond-inputs) from normal inputs that are still within the models' handling capabilities (within-inputs), thus keeping the applications still function with expected reliabilities. The key insight of Dissector is that a DL model should interpret a within-input with increasing confidence, while a beyond-input would probably cause confused guesses in the prediction process. Dissector works in an application-specific way, adaptive to DL models used in applications, and extremely efficiently, scalable to large-size datasets from complex scenarios. The experimental evaluation shows that Dissector outperformed state-of-the-art techniques in the effectiveness (AUC: avg. 0.8935 and up to 0.9894) and efficiency (runtime overhead: only 3.3–5.8 milliseconds). Besides, it also exhibited encouraging usefulness in defensing against adversarial inputs (AUC: avg. 0.9983) and improving a DL model's actual accuracy in use (up to 16% for CIFAR-100 and 20% for ImageNet).

KEYWORDS

deep learning, fault tolerance, input validation

ACM Reference Format:
Huiyan Wang, Jingwei Xu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2020. Dissector: Input Validation for Deep Learning Applications by Crossing-layer Dissection. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Los Alamitos, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380379

1 INTRODUCTION

Deep learning (DL) is assisting applications in a growing way [2, 5, 16, 22, 52, 61, 63]. In such applications, DL models are typically instantiated from training scenarios, and later participate in decision-making as a central classifier for new scenarios. This practice simplifies the software design in specifying how software should behave for complex scenarios. With this benefit, DL applications are increasingly emerging for complex scenarios that are otherwise challenging for traditional programs, e.g., image classification [5], object identification [22], and self-driving [2, 57, 66].

However, due to the difference between new scenarios and training scenarios, as well as the evolution [7] of, and noises in, new scenarios, even a successful DL application can still encounter inputs that are beyond its DL model's handling capability. Then the consequences are unexpected predictions in the decision-making and abnormal behaviors from the application.

One could consider replacing the concerned DL model with a new one (e.g., with different generalizability) [12, 53], or upgrading the model by retraining it with more data from new scenarios (e.g., critical data [56], corner-case data [11, 47, 57], or less-biased

[*] Corresponding authors.

ICSE '20, May 23–29, 2020, Los Alamitos, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380379

data [37]). Although helpful, the replacing or retraining practice is non-trivial, and especially not preferred after the DL application's deployment (e.g., self-driving already put into use). Furthermore, the application could be disrupted from its normal execution, and itself may need rebuilding and redeployment, not to mention that such issues keep arising in future (e.g., new weather conditions, obstacles, or environments encountered on highways).

In this paper, we consider a *fault tolerance* approach. After application deployment, we make the application recognize the inputs that are beyond its DL model's handling capability and prevent them from impacting its decision-making (e.g., isolated or referring to manual driving), since the predictions for these inputs are unexpected and probably misleading or wrong. Then the application is still functional to other inputs and behaves as originally expected, without the need for model retraining or application redeployment. This approach can also be flexible for coping with complex scenarios that can hardly be fully anticipated in advance [2, 16].

Our approach relates to input validation [58, 66] or data cleaning [4, 19–21] efforts, but we argue for addressing the problem from an application's perspective rather than focusing on the inputs themselves only. First, comparing the inputs to original training data may not be feasible, either because of the unavailability or due to the overwhelming runtime overhead [6, 18, 66]. Second, even if one could do so, different DL models can be instantiated from the same training data by different DL algorithms with different parameters, and thus the same validation or cleaning technique can hardly make filtered inputs suit the varying handling capabilities of different DL models. Therefore, our approach has to be *application-specific*, aware of its target DL model (model-aware), as well as being *efficient*, incurring only marginal overhead at runtime.

Besides, our approach has to address a key problem, i.e., *telling when* the inputs to an application are beyond its DL model's handling capability. This is challenging because there is no oracle defining precisely the boundary of a DL model's handling capability, and even the specific DL algorithm used for instantiating this model does not help much on this (its scope of generalization never explicitly specified). Regarding this, we make two observations. First, when the inputs are beyond a DL model's handling capability, their predictions can easily be misleading or wrong, leading to a lower accuracy. Second, such misleading or wrong predictions do not come with no clue; instead, they can be perceived during the DL model's prediction process.

Based on the above analyses, we in this paper propose a novel technique, Dissector, to monitor a DL application's inputs and isolate those possibly beyond its DL model's handling capability from impacting the application's decision-making. First, this technique is model-aware, monitoring whether a DL model interprets an input with increasing confidence in its prediction towards the final result by collecting its uniqueness evidence. If yes, Dissector considers such inputs *within* the model's handling capability (named *within-inputs*), or otherwise, *beyond* that (named *beyond-inputs*). In practice, adversarial inputs that are dedicatedly generated for attacking DL models are one example of beyond-inputs; besides, natural inputs collected by a physical camera (e.g., a photo image) can also be beyond or within according to its target DL model's handling capability. Second, this technique is also lightweight, without any heavy data comparison or analysis operation. This enables it to validate inputs efficiently at runtime, capable for complex scenarios with large-scale data.

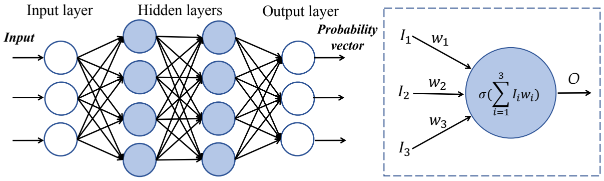


Figure 1: DL model architecture

We experimentally evaluated Dissector and compared it to state-of-the-art techniques (mMutant [58] and Mahalanobis [28]) in distinguishing beyond-inputs from within-inputs. The experiments reported promising results for Dissector (AUC: 0.8223–0.9894), while existing techniques were less effective (AUC: 0.6827–0.9770 and 0.6692–0.8334, respectively). What is worth mentioning is that Dissector applied successfully to large datasets like ImageNet, while existing techniques incurred crashing cases due to large-memory issues. Besides, Dissector was extremely efficient at runtime (3.3–5.8 milliseconds per sample), about 45x and 763x speedup versus existing techniques, respectively. Finally, Dissector also exhibited nice help in defensing against adversarial inputs (AUC: 0.9962–0.9998) and improving a DL model's actual accuracy in use (e.g., by 19% for CIFAR-100 and 20% for ImageNet).

We summarize our contributions in this paper below:

- Proposed a lightweight technique for automated validation of inputs to DL applications (as classifiers).
- Evaluated our technique's performance using real-world large-scale DL models.
- Explored our technique's usefulness in relevant fields.

The remainder of this paper is organized as follows. Section 2 introduces the DL background. Section 3 presents our Dissector technique for automated within-input validation for DL applications. Section 4 experimentally evaluates Dissector's performance and compares it to existing techniques. Section 5 explores Dissector's usefulness in relevant fields. Finally, Section 6 discusses related work in recent years, and Section 7 concludes this paper.

2 BACKGROUND

DL models are used in DL applications mainly for decision-making, which concerns the *model training* and *input predication* the two phases. We below introduce the background on the two phases as well as the related DL model architecture to facilitate our subsequent discussions.

The model training phase takes training data (e.g., a set of image samples, each with a category label like cat or dog), and instantiates (trains) a DL model to represent the knowledge in the data by specific DL algorithms like Convolutional Neural Network (CNN) [12, 53]. Then the input prediction phase takes the trained DL model (or DL model when with no ambiguity) as the knowledge to predict labels for new inputs. To distinguish, we name the inputs used in training and prediction *training samples* and *predicting samples*, respectively.

A DL model is the key architecture in DL, which was inspired from human brains with millions of neurons, transferring information by electrical impulses across layers [24]. DL models are

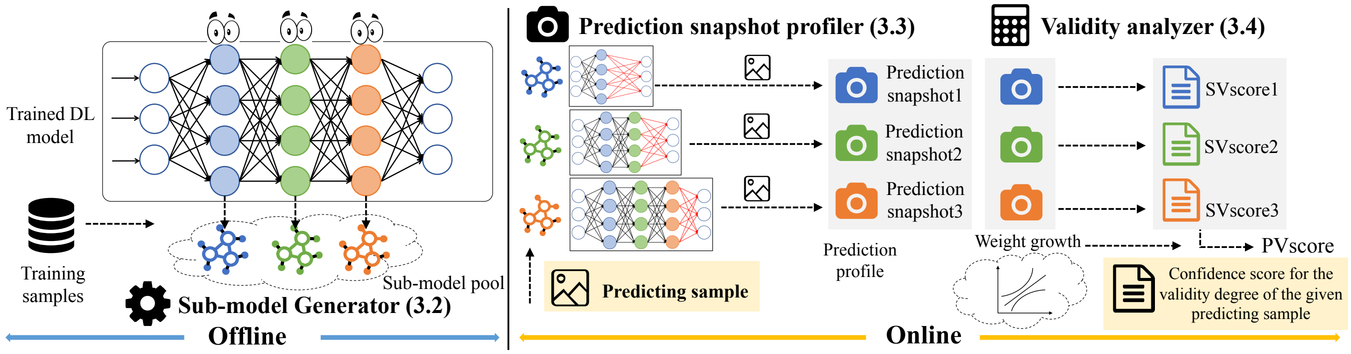


Figure 2: Dissector overview

similarly based on the notion of neurons, which represent features learned from training samples on neurons organized by layers, as shown in Fig. 1 (a fully-connected DNN example). In the model training phase, a DL model learns values for neuron parameters (e.g., bias and weights), which represent features extracted from training samples. Then in the input prediction phase, the parameter values are used for calculating and deciding the most suitable labels for inputted predicting samples. Formally, given a predicting sample, the DL model generates a probability vector in the form of $\{l_0\colon prob_0, l_1\colon prob_1, ..., l_{m-1}\colon prob_{m-1}\}$, suggesting probability $prob_i$ for this sample being classified into label l_i (all probabilities summed up to one if normalized, e.g., by function $softmax$ [43]). Typically, the label with the highest probability is decided as the final prediction result.

3 THE DISSECTOR APPROACH

DL applications rely on their integrated DL models for input prediction and decision-making, and thus their reliability depends largely on these models' prediction accuracies. In this section, we present our Dissector approach to distinguishing beyond-inputs from within-inputs so that the concerned DL applications can work with inputs within their handling capabilities. In the following, we start with an overview of Dissector and then elaborate on its detailed methodology.

3.1 Overview

Dissector consists of three main components, namely, sub-model generator, prediction snapshot profiler, and validity analyzer, as shown in Fig. 2.

The *sub-model generator* (Step 1) takes a trained DL model $\mathcal{M}$ and its corresponding training samples, and returns a sequence of sub-models to represent different levels of knowledge in $\mathcal{M}$. Generally, a sub-model$_k$ encodes the partial knowledge from $\mathcal{M}$'s first to its k-th layers. Note that the sub-model generation is conducted only once in an offline way. Later, the generated sub-models can be reused for validating predicting samples inputted to this DL model.

The *prediction snapshot profiler* and *validity analyzer* work together for online input validation for distinguishing beyond-inputs from within-inputs. Given a predicting sample, the profiler (Step 2) tracks how the sample is interpreted using the sequence of generated sub-models, and makes snapshots at different layers. It composes a final prediction profile based on the collected snapshots. Then the analyzer (Step 3) examines the prediction profile and expects that a within-input should be interpreted by a DL model with increasing confidence towards its final predication result by collecting its uniqueness evidence. Otherwise, it is considered to be a beyond-input. A confidence score (PVscore) is calculated to indicate the likelihood the predicting sample is within the DL model's handling capability. With the score, the application built on this DL model can then decide whether or not to accept this inputted sample and its corresponding prediction result, according to its domain-specific requirements.

We elaborate on the three steps in turn below.

3.2 Step 1: Sub-model Generator

Given a trained DL model $\mathcal{M}$, the first step generates a sequence of sub-models representing the growing partial knowledge encoded in this DL model. These sub-models are used later for validating whether a given input is interpreted by a DL model with increasing confidence towards to its final prediction result.

As illustrated in Fig. 3, each *sub-model*$_k$ is composed of two parts: one part is copied from the original DL model $\mathcal{M}$ (first layer to intermediate layer k) with all structures and parameter values inherited (e.g., neurons, weights, and bias), and the other part is the links from layer k to the output layer (with labels in $\mathcal{M}$), which are newly trained using the original training samples. Generating the first part is straightforward, while generating the second part needs some time, depending on $\mathcal{M}$'s scale, but it is done only once. Here, we adopt a linear regression (LR) structure for the second-part training as the meta-model, since it is one of the most widely used, proved effective structures in DNN for the final prediction layer [8, 12, 26, 53, 55], and has been widely suggested in many existing work [37, 65]. This architecture strictly follows a DL model's typical design. Specifically, the LR structure in a sub-model refers to a one-layer fully connected structure with cross entropy as its loss function.

A DL model can generate as many sub-models as the number of its intermediate layers. Users can, of course, choose to generate all or some, depending on their time budgets. Our later experiments generated only four to seven sub-models, which are already enough

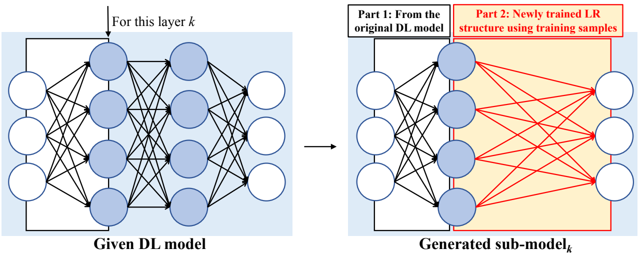

For this layer k
Part 1: From the original DL model
Part 2: Newly trained LR structure using training samples
Given DL model
Generated sub-model$_k$

Figure 3: Sub-model generation

for achieving quite promising results. Besides, we note that our Dissector approach is applicable to general DL models: although we discuss DNN models here, CNN sub-models can be similarly prepared by treating each convolutional layer as a normal layer.

3.3 Step 2: Prediction Snapshot Profiler

When feeding an input to a given DL model, the second step tracks how the DL model interprets this input based on the sub-models generated in the last step. Dissector transforms the problem of validating this input for the original DL model into that of examining how this input is interpreted in these generated sub-models, and obtains a sequence of corresponding probability vectors (normalized by function $softmax$). Since these sub-models represent the growing partial knowledge encoded in the original DL model, these probability vectors explain how the DL model interprets the input layer by layer.

We name each thus obtained probability vector a *prediction snapshot* for the concerned sub-model. By connecting these prediction snapshots in order, Dissector obtains the whole *prediction profile* for this particular input going through all the sub-models, i.e., $\{snapshot_0, ..., snapshot_{n-1}\}$, which is used for examining this input's validity degree in the next step.

3.4 Step 3: Validity Analyzer

Based on the prediction profile obtained for the given input, the third step analyzes the validity for each snapshot in this profile (*snapshot validity*), and then the validity for the whole profile (*profile validity*), so as to calculate the validity degree for the given input.

By validity degree, we expect that a within-input should be predicted in a way that the DL model used for predicting this input should have an increasing confidence when crossing the input layer, through hidden layers, finally to the output layer. This is based on our observation that since a within-input fits in the DL model's handling capability, the model should well recognize this input in its prediction process, instead of being confused by two or more possible guesses during the prediction.

3.4.1 *Snapshot validity measurement.*

For each snapshot in a prediction profile, Dissector measures its validity by analyzing whether and how its corresponding input's final prediction result is *uniquely* supported by the probability vector in this snapshot.

Suppose that an input $\mathcal{I}$ is fed to a DL model $\mathcal{M}$ (with m labels) and $\mathcal{M}$ predicts $\mathcal{I}$ as label l_x. Dissector considers how a snapshot supports this prediction result. Let the snapshot under consideration be $snapshot_k$, which is a probability vector, taking the form of $\{l_0 : prob_0, l_1 : prob_1, ..., l_{m-1} : prob_{m-1}\}$. There are two cases: l_x is already associated with the highest probability in this vector, or otherwise.

For the first case, Dissector (encouragingly) measures the snapshot's unique support by how much l_x's associated probability exceeds the second highest probability in this vector. Let l_{SH} the label having the best shot to confuse the prediction (i.e., with the second highest probability). Intuitively, the larger the difference between the probabilities for label l_x and l_{SH} is, the *more uniquely* the prediction result l_x is supported in this snapshot.

For the second case, l_x is not associated with the highest probability. Dissector (penalizingly) measures the snapshot's unique support by how much the actual highest probability (with label l_H) exceeds that of l_x in this vector. Similarly, the larger the difference between the probabilities for label l_H and l_x is, the *less uniquely* the prediction result l_x is supported in this snapshot.

Following this intuition, we measure the snapshot validity as follows (let $snapshot[l]$ return l's associated probability):

$$SVscore_k(l_x, snapshot_k) =$$
$$\begin{cases} \frac{snapshot_k[l_x]}{snapshot_k[l_x]+snapshot_k[l_{SH}]}, & l_x \text{ with the highest probability;} \\ 1 - \frac{snapshot_k[l_H]}{snapshot_k[l_x]+snapshot_k[l_H]}, & \text{otherwise.} \end{cases}$$

The snapshot validity score falls in range [0, 1]. The closer it is to 1, the more uniquely the current snapshot supports the final prediction result l_x.

3.4.2 *Profile validity measurement.*

Based on the sequence of calculated snapshot validity scores, Dissector then measures the validity for the whole prediction profile with respect to our expected increasing confidence for a DL model in interpreting within-inputs towards their final prediction results.

Since each snapshot corresponds to one particular intermediate layer in the original DL model $\mathcal{M}$, we normalize these snapshot validity scores with increasing weights from the first intermediate layer to the last one (i.e., increasing k), echoing our aforementioned "increasing confidence". Therefore, we measure the profile validity as follows:

$$PVscore(l_x) = \frac{\sum_{k=1}^{n} SVscore_k(l_x, snapshot_k) \times weight_k}{\sum_{k=1}^{n} weight_k}.$$

3.4.3 *Weight parameter setup.*

The preceding equation requires the setup for a sequence of increasing weight values. Instead of giving ad hoc choices or tuning specially for our experimental subjects, we try three popular growth types, namely, *linear*, *logarithmic*, and *exponential*, in our Dissector framework.

Table 1 lists general formulas for calculating weight values with respect to these three growth types, where x corresponds to number k of the specific layer ($snapshot_k$) and y represents its corresponding weight value $weight_k$. However, these formulas contain too many parameters for setup. Fortunately, they are subject to reduction without losing essentials, since the weight values participate in both numerators and denominators of the equation. As shown in Table 1, the reduced formulas contain one (α) or two (α and β) parameters only, and in our later evaluation we would investigate the impacts of their values on Dissector's performance.

Table 1: Parameter reduction for modeling weights

Growth type	General formula	Reduced formulas	# para
Linear	$y = ax + k$	(1) $y = x$ (2) $y = \alpha x + 1$	2 → 1
Logarithmic	$y = a\log_b(kx + c_1) + c_2$	(1) $y = \ln x$ (2) $y = \alpha \ln x + 1$ (3) $y = \alpha \ln(\beta x + 1) + 1$	5 → 2
Exponential	$y = ae^{kx+b_1} + b_2$	(1) $y = e^{\beta x}$ (2) $y = \alpha e^{\beta x} + 1$	4 → 2

Generally, for a given input $\mathcal{I}$ and a DL model $\mathcal{M}$, Dissector's calculated $PVscore$ value represents $\mathcal{I}$'s validity degree with respect to $\mathcal{M}$, and this value has been normalized to [0, 1]. The closer the $PVscore$ value is to one, $\mathcal{I}$ is more like a within-input to $\mathcal{M}$ (within $\mathcal{M}$'s handling capability), or otherwise, more like a beyond-input to $\mathcal{M}$ (beyond $\mathcal{M}$'s handling capability).

4 EVALUATION

In this section, we evaluate Dissector and compare it to existing techniques in distinguishing beyond-inputs from within-inputs for DL applications.

4.1 Research Questions

We aim to answer the following four research questions:

RQ1 (Effectiveness): *How effective is Dissector in distinguishing beyond-inputs from within-inputs, as compared to existing techniques?*

RQ2 (Efficiency): *How efficient is Dissector in this process, as compared to existing techniques?*

RQ3 (Controlling factors): *How do Dissector's model-aware treatment, weight growth type, and internal parameters affect its effectiveness?*

RQ4 (Usefulness): *How does Dissector help defense against adversarial inputs and improve a DL model's actual accuracy in use?*

4.2 Experimental Subjects, Design, and Process

We introduce experimental subjects, design, and process below.

Experimental subjects. We used as experimental subjects four popular image classification datasets in the DL field, namely, MNIST, CIFAR-10, CIFAR-100, and ImageNet, each associated with a state-of-the-art DL model, as shown in Table 2. (1) *MNIST* [25] is an image database for hand-writing digit classification (ten labels). It contains 60,000 training samples and 10,000 predicting samples for testing. Its DL model is LeNet4 [26]. (2) *CIFAR-10* [22] is an image database for object recognition (also ten labels). It contains 50,000 training samples and 10,000 predicting samples. Its DL model is WRN-28-10 (WRN for short) [64]. (3) *CIFAR-100* [22] is similar to CIFAR-10 but with 100 labels. It contains 50,000 training samples and 10,000 predicting samples. Its DL model is ResNeXt-29 (8x64d) (ResNeXt for short) [60]. (4) *ImageNet* [5] is a much larger database for image recognition (with 1,000 labels). It contains 1.2 million training samples and 50,000 predicting samples. We used its ILSVRC2012 version [51], and its DL model is ResNet101 [12].

Experimental design. We designed the following two independent variables to control the experiments:

Table 2: Descriptions for datasets and associated DL models

Dataset	Description	# labels	# samples	DL model
MNIST	Digit classification	10	60,000/10,000	LeNet4
CIFAR-10	Object recognition	10	50,000/10,000	WRN
CIFAR-100	Object recognition	100	50,000/10,000	ResNeXt
ImageNet	Image recognition	1,000	1,200,000/50,000	ResNet101

- *Subject.* We used a total of four subjects, each concerning a dataset and a DL model, namely, MNIST+LeNet4, CIFAR-10+WRN, CIFAR-100+ResNeXt, and ImageNet+ResNet101. When with no ambiguity, we refer to each one by the dataset name only.
- *Technique.* We compared Dissector with two state-of-the-art techniques closely related to input validation, namely, mMutant [58], based on model mutation analysis, and Mahalanobis [28], based on data-distance measurement. Dissector was configured into three variants (three weight growth types), namely, *Dissector-linear*, *Dissector-log*, and *Dissector-exp*. mMutant was configured into four variants (four mutation operators), namely, *mMutant-GF*, *mMutant-NAI*, *mMutant-WS*, and *mMutant-NS* (no mixture recommended [58]). *Mahalanobis* was configured using its original setting [28]. All these techniques were either originally designed or slightly adapted to report a normalized score in [0, 1] for a given input, with a value close to 1 suggesting more "within" and to 0 more "beyond".

The three techniques need some setups. (1) Dissector needs to select DL model layers for sub-model generation. For LeNet4, we selected all layers since it has only four. For complex models, we selected part of their layers for efficiency. For WRN and ResNext, we selected layers after each block, and for ResNet101, we selected layers uniformly within each convolution group (i.e., layers 10, 22, 46, 70, 91, 100, and 101). (2) mMutant needs to configure a mutation degree. This parameter was set to 0.05 for MNIST, and 0.005 for CIFAR-10 as suggested [58]. It was also set to 0.005 for CIFAR-100 as it is similar to CIFAR-10. However, mMutant failed to apply to ImageNet due to its huge model sizes (incurring OutOfMemoryException (OOM)). Even if one could do so, we estimated that it needs 1,500–2,000 mutants for ImageNet, which would cost over 1,000GB RAM and significant time overhead (over weeks with our GPU resources). So we had to give it up. (3) For Mahalanobis, we followed its original setting [28] to select 10% samples under test to train its weight parameters, and thus its effectiveness measurement was based on remaining 90% samples. Similarly, Mahalanobis also failed to apply to ImageNet, causing OOM exceptions when its GDA classifier analyses stored and processed intermediate results concerning training samples.

We designed the following two dependent variables to evaluate the three techniques:

- *AUC.* We used the popular Area Under Curve (AUC) based on True Positive Rate (TPR) and False Positive Rate (FPR) data to measure how effective a technique is in distinguishing beyond-inputs from within-ones. To do so, by varying a threshold from 0 to 1 and comparing it to the technique's reported scores for its received inputs, these inputs can be separated into a corresponding set of within-ones and that

Table 3: PVscore comparison between incorrectly-predicted samples and correctly-predicted samples

Subject (dataset + DL model)	Accuracy	Incorrectly-predicted samples				Correctly-predicted samples				t-test (indep.)	
		# samples	Avg.	Std. v.	Med.	# samples	Avg.	Std. v.	Med.	t value	p value
MNIST + LeNet4	98.41%	159	0.52	0.18	0.49	9,841	0.96	0.06	0.98	30.10	1.85e-67
CIFAR-10 + WRN	96.21%	379	0.75	0.16	0.78	9,621	0.94	0.08	0.98	22.46	4.44e-72
CIFAR-100 + ResNeXt	82.15%	1,785	0.61	0.17	0.61	8,215	0.84	0.14	0.88	53.29	≈ 0.0[1]
ImageNet + ResNet101	77.31%	11,344	0.49	0.22	0.47	38,656	0.76	0.19	0.81	121.59	≈ 0.0

[1] denoting that the corresponding p value is quite close to 0.0, with the difference even smaller than sys.float_info.epsilon in python.

of beyond-ones (both varying). Then by comparing the two sets of inputs to the ground truths of real within-inputs and beyond-inputs (discussed later), one can calculate respective TPR and FPR data and corresponding AUC values (areas below the curves): the larger, the more effective the technique is in distinguishing beyond-inputs from within-ones.

- *Time overhead.* We recorded the time spent by a technique on its offline work (offline overhead) and online work (online overhead). The offline work is the sub-model preparation for Dissector, mutant generation for mMutant, and layer-wise GDA classifier analysis for Mahalanobis. The online work is the validity analysis for Dissector, LCR analysis for mMutant, and distance scoring for Mahalanobis.

Experimental process. We conducted all experiments on a Linux server with three Intel Xeon E5-2660 v3 CPUs @2.60GHz, 16 Tesla K80 GPUs, and 500GB RAM, running Ubuntu 14.04.

For RQ1 (effectiveness), we calculate AUC values to compare Dissector with mMutant and Mahalanobis for their effectiveness in distinguishing beyond-inputs from within-inputs.

For RQ2 (efficiency), we measure offline and online time overheads to compare Dissector with mMutant and Mahalanobis for their feasibilities in the input validation. The offline time overhead (once) should be acceptable, and the online time overhead (repeated) should be marginal.

For RQ3 (controlling factors), we study how Dissector's effectiveness can be affected by its internal factors.

For RQ4 (usefulness), we study how Dissector can help defense against adversarial inputs and improve a DL model's actual accuracy in use. For the former, we composed a set of inputs that contain both training samples and adversarial ones generated from them by a popular attacker FGSM [10]. We study how effective Dissector is in distinguishing these two types of samples. For the latter, we controlled the threshold to study how Dissector improves a DL model's actual accuracy in use. We also calculated the number of samples thus isolated, which should be acceptable.

4.3 Experimental Results and Analyses

We report and analyze experimental results, and answer the preceding four research questions in turn.

4.3.1 *RQ1 (Effectiveness).*

As mentioned earlier, calculating AUC values for effectiveness measurement needs the ground truths of real beyond-inputs and within-inputs. In the following, we first discuss a candidate for simulating the ground truths and then calculate AUC values based on it.

Ground truths. We note that the ground truths of real beyond-inputs and within-inputs do not naturally exist, because otherwise distinguishing beyond-inputs from within-inputs becomes a trivial thing and already solved. Regarding this, we consider finding a candidate for simulating the ground truths. According to earlier discussions, beyond-inputs probably cause a DL model to predict in a misleading or wrong way. For example, for the four subjects, a random guess would cause an accuracy of 10%, 10%, 1%, and 0.1%, far below their DL models' accuracies (77–98%, as in Table 3). Therefore, we consider a proper candidate for beyond-inputs like those incorrectly-predicted samples, while that for within-inputs like those correctly-predicted samples. Furthermore, if this simulation is reasonable, Dissector should be able to distinguish them clearly by its PVscore measurement. Therefore, we study the two sets of samples in TableTable 3 to see whether they behave as expected.

From the table, we observe that: (1) the incorrectly-predicted samples from all predicting samples obtained only a 0.49–0.75 PVscore value (avg. 0.59), while those correctly-predicted samples obtained 0.76–0.96 (avg. 0.88), with clear differences; (2) for each subject, its difference is still evident, e.g., 0.52 vs. 0.96, 0.75 vs. 0.94, 0.61 vs. 0.84, and 0.49 vs. 0.76, with the latter close to its model accuracy.

We used t-test [38] to measure how significant such differences are in a statistic way. With the null hypothesis that "Dissector generated PVscore values with no significant difference between incorrectly-predicted and correctly-predicted samples", we obtained a series of p-values listed in the last column of Table 3, all of which are far less than 0.05. Thus one can reject this hypothesis at a 95% confidence level. With this result, we would later use the two sets of incorrectly-predicted samples and correctly-predicted samples for each subject to simulate the ground truths of real beyond-inputs and within-inputs to facilitate the effectiveness comparison. The point is that although this simulation is a rough estimation, the logic still holds (from being a beyond-input to being predicted probably incorrectly), which suffices for our experimental comparisons.

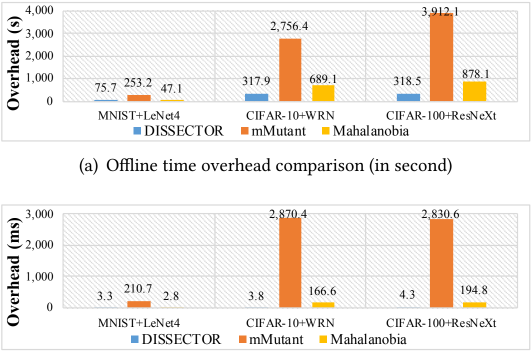
AUC comparison. As mentioned earlier, we measure AUC values to compare the three techniques' effectiveness in distinguishing beyond-inputs from within-inputs. For Dissector's three variants, we first studied their simplest forms, i.e., Dissector-linear ($y = x$), Dissector-log ($y = \ln x$), and Dissector-exp ($y = e^x$). Table 4 lists AUC comparison results (the larger, the better, with range [0, 1]).

From the table, we observe that: (1) for each of the four subjects, Dissector always obtained the best AUC values, e.g., 0.9869–0.9894 for MNIST, 0.8740–0.8963 for CIFAR-10, 0.8516–0.8726 for CIFAR-100, and 0.8223–0.8562 for ImageNet; (2) mMutant obtained only 0.9449–0.9770, 0.7346–0.8643, and 0.6827–0.7129 for the first three

Table 4: AUC comparison among three techniques in distinguishing beyond-inputs from within-inputs

Technique	MNIST +LeNet4	CIFAR-10 +WRN	CIFAR-100 +ResNeXt	ImageNet +ResNet101
Dissector-linear	**0.9894**	0.8740	0.8516	0.8250
Dissector-log	0.9869	**0.8963**	0.8641	0.8223
Dissector-exp	0.9878	0.8960	**0.8726**	**0.8562**
mMutant-GF	0.9712	0.8643	0.6999	OOM
mMutant-NAI	0.9770	0.8577	0.7129	OOM
mMutant-WS	0.9449	0.8483	0.6827	OOM
mMutant-NS	0.9575	0.7346	0.6871	OOM
Mahalanobis	0.7504	0.8334	0.6692	OOM

the highest AUC value for each subject is made bold.



(a) Offline time overhead comparison (in second)

(b) Online time overhead comparison (in millisecond)

Figure 4: Time overhead comparison

subjects (with clear gaps to Dissector), and crashed with OOM exceptions for the largest subject ImageNet; (3) Mahalanobis obtained even lower values, 0.7504, 0.8334, 0.6692, for the first three subjects, respectively, and also failed for the largest subject ImageNet; (4) the three Dissector variants behaved similarly satisfactorily, and Dissector-exp worked slightly better for ImageNet; (5) for large subjects like CIFAR-100 and ImageNet, Dissector behaved more stably (still above 0.85 and 0.82), but mMutant and Mahalanobis either led to largely reduced AUC values (below 0.72 and 0.67) or crashed with OOM exceptions.

Therefore, we answer RQ1 as follows: *Dissector was effective in distinguishing beyond-inputs from within-inputs (AUC: 0.8223–0.9894), and behaved more stably and suitably than existing techniques for subjects of varying accuracies and sizes.*

4.3.2 RQ2 (Efficiency). We then compare the three techniques' time overheads, for both offline and online work. We reported the comparison data for the first three subjects in Fig. 4 (ImageNet data were either incomplete or too large to fit in the figure, discussed later). For Dissector, we chose Dissector-linear ($y = x$), and for mMutant, we chose mMutant-NAI (other variants were similar to the chosen ones in time overheads).

From Fig. 4(a) and Fig. 4(b), we observe that: (1) regarding the offline overhead, mMutant always took the most time: 253.2 seconds for MNIST, 2,756.4 seconds for CIFAR-10, and 3,912.1 seconds for CIFAR-100, which are significantly more than what Dissector and Mahalanobis cost: 75.7, 317.9 (least), and 318.5 (least) seconds for the former, and 47.1 (least), 689.1, and 878.1 seconds for the latter; (2) regarding the online overhead, mMutant again took the most time: 210.7 milliseconds per sample for MNIST, 2,870.4 milliseconds for CIFAR-10, and 2,830.6 milliseconds for CIFAR-100, which are also significantly more than what Dissector and Mahalanobis cost: 3.3, 3.8 (least), and 4.3 (least) milliseconds per sample for the former, and 2.8 (least), 166.6, and 194.8 milliseconds for the latter; (3) altogether, Dissector took only several minutes to complete its offline preparation, and several milliseconds to validate an input at runtime, which are very attractive; (4) for the more important runtime input validation, mMutant had to take 63.3–763.3x time to that of Dissector, and Mahalanobis took 0.8–45.3x time.

Table 5: AUC comparison for studying Dissector's model-aware treatment

Analyzing scenario	Application scenario		
(For MNIST)	**LeNet4**	**DNN2**	**LeNet5**
LeNet4 [26]	**0.9878**	0.9005 (−7.3%)	0.9574 (−3.1%)
DNN2[1]	0.9129 (−7.6%)	**0.9716**	0.9063 (−8.3%)
LeNet5 [26]	0.9720 (−1.6%)	0.9248 (−4.8%)	**0.9882**
(For CIFAR-10)	**WRN**	**VGG16**	**DenseNet**
WRN [64]	**0.8960**	0.8695 (−5.2%)	0.8828 (−2.2%)
VGG16 [53]	0.8837 (−1.4%)	**0.9176**	0.8759 (−2.9%)
DenseNet [8]	0.8686 (−3.1%)	0.8626 (−6.0%)	**0.9024**
(For CIFAR-100)	**ResNeXt**	**VGG16**	**DenseNet**
ResNeXt [60]	**0.8726**	0.8352 (−3.3%)	0.8425 (−2.4%)
VGG16 [53]	0.7680 (−12.0%)	**0.8636**	0.7798 (−9.7%)
DenseNet [8]	0.7980 (−8.5%)	0.7997 (−7.4%)	**0.8634**
(For ImageNet)	**ResNet101**	**ResNet50**	**VGG16**
ResNet101 [12]	**0.8562**	0.8432 (−2.1%)	0.8408 (−0.9%)
ResNet50 [12]	0.8308 (−3.0%)	**0.8609**	0.8440 (−0.5%)
VGG16 [53]	0.7938 (−7.3%)	0.8012 (−6.9%)	**0.8483**

[1] denoting a simple two-hidden-layer fully connected multilayer neural network.

For the largest subject ImageNet with its ResNet101 model, Dissector spent around 90 hours on its offline preparation, and needed 5.8 milliseconds for its online validation per sample. 90 hours are comparable to the subject' own DL model' training time [62]. Since required only once, the time is acceptable. Dissector's online overhead is still marginal (several milliseconds), suggesting that it is extremely stable even for complex subjects like ImageNet. On the contrary, both mMutant and Mahalanobis failed to apply to this subject as explained earlier.

Therefore, we answer RQ2 as follows: *Dissector was highly efficient with acceptable offline overhead and marginal online overhead (several milliseconds), and much faster than existing techniques (up to 12.3x and to 763.3x speedup for offline and online work, respectively).*

4.3.3 RQ3 (Controlling factors). We next study how Dissector's model-aware treatment (validating inputs with respect to DL models used in applications) contributes to its effectiveness, and how its inherent growth type and other parameters affect the effectiveness.

Model-aware treatment. Dissector validates inputs and identifies beyond-inputs based on its prepared sub-models, which are derived from DL models used in applications. Thus, Dissector is naturally model-aware, and we have observed Dissector's unique effectiveness from this treatment earlier. Still, we want to know how Dissector's effectiveness would be compromised if the model its analysis depends on deviates from the model it is applied to. To be specific, the former model is the one from which Dissector prepares sub-models, and the latter model is the one that defines/simulates the sets of real beyond- and within-inputs (i.e., ground truths). If Dissector's model-aware treatment is not necessary, two models being different will not cause the effectiveness largely compromised; otherwise, it will.

For this part of experiments, we took the best Dissector-exp for example. Besides, for each subject, we additionally associated it with two more popular DL models, and thus each dataset was now with three DL models, as in Table 5 (original DL model is listed at the first row and first column). Note that these new DL models may have different accuracies, e.g., for CIFAR-100, the three models had an accuracy of 82.15%, 68.61%, and 73.97% respectively, exhibiting the required diversity for experiments.

Table 5 compares AUC values among nine combinations for each of the four subjects, in which the *analyzing scenario* refers to the model Dissector's analysis depends on and the *application scenario* refers to the model it is applied to. From the table, we observe that: (1) when the analyzing and application scenarios were the same, Dissector indeed always obtained the best effectiveness results (diagonal values in each rectangle area, marked in bold); (2) when the two scenarios were different, AUC values were compromised by a varying degree of 0.5–12.0%, which is not small; (3) in spite of scenarios being different, Dissector could still obtain mostly higher AUC values than existing techniques for scenarios being the same, e.g., better than the best mMutant-NAI in 67.5% cases, and better than Mahalanobis in 100% cases.

Therefore, we consider Dissector's model-aware treatment necessary and important for its best effectiveness. Besides, even if it is compromised, Dissector could still bring satisfactory results.

Weight growth type and other parameters. As mentioned earlier, Dissector can be configured into three variants (with three weight growth types, namely, logarithmic, linear, and exponential). Besides, it could be further customized by two parameters, α and β. Previously, we explored only the simplest forms for the three variants, and now we study the impact of different α and β values (from 1 to 100).

Table 6 compares the impact of Dissector's weight growth type and α and β parameters on its three variant families, according to the setup in Table 1. From the table, we observe that: (1) three weight growth types behave similarly with stable AUC values for each subject (0.0000–0.0085, 0.0000–0.0446, and 0.0000–0.0459 differences, respectively); (2) although the exponential growth could be a bit unstable for some cases, it obtained the best results mostly, with AUC values highest (0.9377, 0.8855, and 0.8564) for CIFAR-10+WRN, CIFAR-100+ResNeXt, ImageNet+ResNet101, respectively, and quite close (only 0.0018 gap) to the highest AUC value (0.9900) for MNIST+LeNet4, and we owe the results to this type's nature (e.g., more aggressive in the growth and validation); (3) AUC values are all over 0.80, with most (around 76%) over 0.85 and up to 0.9900, suggesting Dissector's general effectiveness among a wide range of parameter values.

Table 6: AUC comparison for the impact of weight growth type and other parameters

Growth type	(α, β)	MNIST +LeNet4	CIFAR-10 +WRN	CIFAR-100 +ResNeXt	ImageNet +ResNet101
Linear	$y = x$	0.9894	0.8740	0.8516	0.8250
	(1,-)	0.9897	0.8650	0.8431	0.8250
	(10,-)	0.9894	0.8726	0.8505	0.8241
	(100,-)	0.9894	0.8738	0.8515	0.8249
Logarith -mic	$y = \ln x$	0.9869	0.8963	0.8641	0.8223
	(1,-)	0.9898	0.8657	0.8411	0.8147
	(10,-)	0.9880	0.8894	0.8598	0.8212
	(100,-)	0.9871	0.8953	0.8636	0.8222
	(1,1)	0.9899	0.8556	0.8414	0.8110
	(1,10)	0.9900	0.8534	0.8287	0.8086
	(1,100)	0.9900	0.8508	0.8258	0.8067
	(10,1)	0.9898	0.8632	0.8393	0.8152
	(10,10)	0.9899	0.8557	0.8311	0.8100
	(10,100)	0.9900	0.8517	0.8268	0.8073
	(100,1)	0.9898	0.8645	0.8308	0.8158
	(100,10)	0.9899	0.8560	0.8315	0.8107
	(100,100)	0.9900	0.8518	0.8269	0.8074
Exponen -tial	$y = e^x$	0.9878	0.8960	0.8726	0.8562
	(-,1)	0.9878	0.8960	0.8726	0.8562
	(-,10)	0.9768	0.9377	0.8855	0.8564
	(-,100)	0.9768	0.9377	0.8855	0.8564
	(1,1)	0.9882	0.8918	0.8705	0.8561
	(1,10)	0.9768	0.9377	0.8855	0.8564
	(1,100)	0.9768	0.9377	0.8855	0.8564
	(10,1)	0.9878	0.8955	0.8724	0.8564
	(10,10)	0.9768	0.9377	0.8855	0.8564
	(10,100)	0.9768	0.9377	0.8855	0.8564
	(100,1)	0.9878	0.8959	0.8726	0.8562
	(100,10)	0.9768	0.9377	0.8855	0.8564
	(100,100)	0.9768	0.9377	0.8855	0.8564

The exponential growth seems to more favor complex subjects (i.e., the latter three subjects). It might be due to the concerned DL models' structures. For the first subject MNIST+LeNet4, its model is of relatively simple structure (only four layers), and the linear and logarithmic growths can well model its weight with normal growth types. For the latter three complex subjects, their models are of quite complex structures (e.g., ResNet101 with around one hundred layers), thus incurring quite complex behaviors in layer-wise sample predictions, and the exponential growth can better model their weights with own aggressive growth.

Therefore, we answer RQ3 as follows: *Dissector's model-aware treatment is necessary and important for its best effectiveness; besides, its weight growth type and α and β parameters slightly affect its stableness, but its effectiveness generally holds.*

4.3.4 *RQ4 (Usefulness).*

We finally study how Dissector helps by its input validation in *defensing* against adversarial inputs and improving a DL model's actual accuracy in use.

Defensing against adversarial inputs. We are interested in whether Dissector can also identify adversarial attacking samples by its beyond-input recognition. As mentioned earlier, we used a popular adversarial attacker FGSM [10] to generate adversarial attacking samples (L_2 norm adopted and attack step set to 0.016 [23]) based on original predicting samples from the four subjects. *Clean samples* were selected from the predicting samples when their predictions were correct, and *adversarial attacking samples* were selected from the generated ones when their predictions were

Table 7: AUC comparison among three techniques in identifying adversarial attacking samples

Technique	MNIST +LeNet4	CIFAR-10 +WRN	CIFAR-100 +ResNeXt	ImageNet +ResNet101
Dissector-linear	0.9979	0.9996	0.9979	0.9966
Dissector-log	0.9980	0.9997	0.9981	0.9962
Dissector-exp	**0.9987**	**0.9998**	**0.9990**	**0.9986**
mMutant-GF	0.9665	0.7792	0.7998	OOM
mMutant-NAI	0.9752	0.7637	0.7652	OOM
mMutant-WS	0.9441	0.7952	0.7715	OOM
mMutant-NS	0.9557	0.7478	0.7739	OOM
Mahalanobis	0.8152	0.9276	0.9949	OOM

the highest AUC value for each subject is made bold.

incorrect. Then we study whether Dissector can distinguish adversarial attacking samples from clean samples as it did for beyond- and within-inputs in earlier experiments.

Table 7 lists Dissector's AUC values on this aspect, and also compares it to earlier techniques mMutant and Mahalanobis. From the table, we observe that: (1) when mixing adversarial attacking samples and clean samples together, Dissector obtained highest and stable AUC values (always over 0.9962 and up to 0.9998) in identifying the former from them, outperforming the other techniques (0.7478–0.9752 for mMutant and 0.8152–0.9949 for Mahalanobis) for their applicable subjects; (2) mMutant was unstable with AUC values ranging in 0.7478–0.9752, performing no better than any Dissector variant (Dissector-exp won in all cases), and failed for ImageNet due to OOM exceptions; (3) Mahalanobis obtained better AUC values than mMutant for CIFAR-10 and CIFAR-100, but behaved worse for MNIST and similarly failed for ImageNet.

As a whole, we consider Dissector very useful (AUC value nearly one) in identifying adversarial attacking examples by beyond-inputs, as one of popular DL-based security applications. We owe this to Dissector's dedicated design of examining increasing confidence for given inputs, thus able to identify adversarial ones by profiling and analyzing their whole crossing-layer prediction process, since a traditional attack can seldom consider the whole DL model in a crossing-layer way.

Improving a DL model's actual accuracy in use. Each DL model is associated with an accuracy when given a set of predicting samples for testing (e.g., those listed in Table 3). With Dissector, the given predicting samples are refined by isolating those beyond-inputs, and then the remaining ones (within-inputs) can bring a better accuracy. To distinguish, we name the former *original accuracy* and the latter *actual accuracy in use*. We are interested in how much Dissector can help improve the accuracy. Note that this is just one possible application of Dissector (more discussed later).

We use a threshold to decide whether a predicting sample with a distinct PVscore value reported by Dissector is a within- (when above) or beyond- (when below) input. Fig. 5 illustrates how a DL model's actual accuracy was improved under different thresholds. From the figure, we observe that: (1) all three Dissector variants (linear, logarithmic, and exponential growths with simplest forms) exhibited clear accuracy improvements with the growth of Dissector's threshold value; (2) for subjects MNIST and CIFAR-10, whose original accuracies were already very high (98.41% and 96.21%), Dissector's contributions to their accuracy improvement were relatively slow but steady (finally reaching 99–100%); (3) for subjects CIFAR-100 and ImageNet, whose original accuracies were somewhat low (82.15% and 77.31%), Dissector's contributions to their accuracy improvement were quite impressive (finally by 15–16% and 19–20%); (4) even with a conservative threshold value of 0.8, the three Dissector variants realized a satisfactory actual accuracy of 95–100%, 94–100%, and 91–100%, respectively.

Some applications might concern the number of isolated samples as the cost for the accuracy improvement. So we also investigated this issue. Take Dissector-linear ($y = x$) for example. When one set the threshold to 0.6 and 0.8, the four subjects' actual accuracies could already be largely improved, e.g., CIFAR-100's accuracy improved from 82% to 89% and 96%, respectively. Accordingly, its number of isolated samples was 14% and 43%. If it was Dissector-exp, the number of isolated samples was much less, e.g., 9% and 21%, respectively. Nevertheless, the subject itself also played an important role on this number, e.g., for MNIST (with the highest original accuracy), isolated samples were only 0–1% for both thresholds and all Dissector variants. As a comparison, when using other techniques for input validation, we encountered more isolated samples. For example, when expecting the same 89% actual accuracy for the CIFAR-100+ResNeXt subject, mMutant-NAI and Mahalanobis caused 21% and 47% samples being isolated but Dissector-exp isolated only 9% samples. This suggests that Dissector's beyond-input identification is more precise and thus more cost-effective.

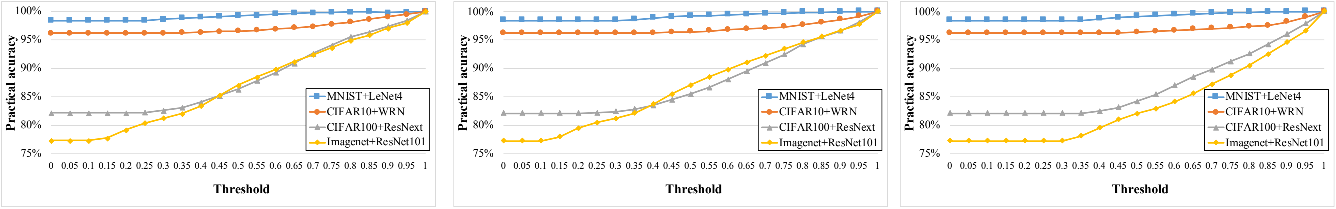
This application does not have to retrain the concerned DL model, and instead brings a transparent accuracy improvement, thus being very attractive. The fault tolerance idea behind Dissector actually maximizes the potential of the original DL model about what it can actually do. Besides the simple treatment of isolating the identified beyond-inputs, one can also refer to other DL models, applications, or even manual controls (e.g., for self-driving). This direction deserves more research efforts.

Therefore, we answer RQ4 as follows: *Dissector are useful for defensing against adversarial inputs and improving a DL model's actual accuracy in use.*

4.4 Threat Analyses

Our selection of the four subjects, namely, MNIST, CIFAR-10, CIFAR-100, and ImageNet with their associated DL models, might threaten the *external validity* of our experimental conclusions. We tried to alleviate this threat by the following efforts: (1) the four datasets are very popular and have been widely used in relevant research [11, 18, 47, 58]; (2) their associated models are also state-of-the-art DL ones; (3) these datasets and models differ from each other by varying topics (e.g., digit, image, and general object recognitions), labels (from 10 to 1,000), scales (from 70,000 to 1,250,000 samples), model types (e.g., LeNet4, WRN, ResNeXt, and ResNet101), model layers (from 5 to 101 layers), and model accuracies (from 77.31% to 98.41%), which make these subjects diverse and representative. Therefore, our experimental conclusions should generally hold, although specific data could be inevitably different for other subjects.

Threats to the *external validity* might also come from our selected techniques for the experimental comparisons, which include



(a) Dissector-linear

(b) Dissector-log

(c) Dissector-exp

Figure 5: Accuracy improvement under different thresholds for Dissector

mMutant [58] and Mahalanobis [28]. We note that as we argued earlier, identifying beyond-inputs for a DL application should be from the perspective of the application and ought to be lightweight runtime validation. Thus, existing work right focused on this problem and meeting the restriction can be little (excluding some options, e.g., DeepRoad [66] and Surprise [18]). We selected mMutant because it resembles our focus by identifying unqualified inputs by DL model mutation analysis (software engineering area). Besides, the work was published in May 2019, as the representative of the state-of-the-art techniques. We also selected Mahalanobis because it similarly uses layer-wise information to identify out-of-distribution samples for DL models based on distance measurements (artificial intelligence area). The work is also the latest (Dec 2018), closely related to our problem, as the representative of distance-based and distribution-based data comparison techniques.

Our *internal threat* mainly comes from the lack of ground truths for distinguishing beyond-inputs from within-inputs. We used the inputs predicted incorrectly and those predicted correctly to simulate beyond-inputs and within-inputs, respectively. As discussed earlier, such estimation might be rough, but since the logic (from beyond-input to probably incorrect prediction) holds (RQ1), our experimental conclusions would largely hold. To further alleviate this threat, we conducted the experiments to study Dissector's usefulness (RQ4), which frankly disclosed what Dissector can indeed help defense against adversarial inputs and improve a DL model's actual accuracy in use, even if based on our ground truth simulation. Considering that our subjects have necessary diversities as discussed above, our experimental conclusions can hold generally. Still, due to the ground truth problem, we plan to validate Dissector in more and practical application scenarios.

5 DISSECTOR APPLICATIONS

We discuss potential Dissector applications below:

Tolerating imperfect DL models. DL models can hardly be perfect for complex application scenarios. Even if they are acceptable for now, application scenarios can evolve from time to time and cause the models to behave unexpectedly unsatisfactorily, as discussed earlier. With an input validation wrapper like Dissector, a DL application built on such DL models can be substantially improved by automatically recognizing beyond-inputs with respect to what these DL models actually do. Especially when the application is deployed in an unstable environment, such an automated technique does help in a convenient way.

Refining DL models. For the case DL models themselves have to be refined, Dissector's identified beyond-inputs form a critical set for consideration. This set of inputs are beyond a DL model's handling capability, and would probably cause misleading or incorrect predictions. Then users can consider whether and how to use them, e.g., for expanding the model's scope by retraining it with these beyond-inputs, or strengthening its original scope by keeping them isolated. More issues such as model stability and corner cases can also be taken into consideration during this refinement process.

Comparing DL models' accuracies. Traditionally, DL models can be directly compared by their prediction accuracies with respect to given samples from an application scenario. With a Dissector-alike input validation wrapper, the comparison can now have new considerations. Suppose that models A and B have original accuracies of 75% and 80% for this scenario. With Dissector, their actual accuracies in use might be improved to 90% and 85% instead. This calls for new research opportunities on how to understand a DL model's actual accuracy in practice.

Measuring deployment suitability. A DL application might be deployed in a complex application scenario, which cannot be fully anticipated or tested. Based on how many inputs are identified as beyond-ones as well as the resulting accuracy, Dissector can be used for suggesting whether and how the concerned DL model suits its deployment. More applications include assigning the most suitable DL model from a set of candidates to the scenario, as well as balancing the assignment of multiple DL models to multiple application scenarios.

Defensing against adversarial inputs. Adversarial inputs can be substantial attacks to a DL application, and thus identifying them is beneficial. Our evaluation shows that although most adversarial inputs behaved like beyond-inputs, few of them might still pass the validation and behave like within-ones. Currently, there is strong evidence [1] showing that taking all adversarial inputs into retraining could probably bias a DL model. Then this set of adversarial, yet still within-inputs, becomes an interesting source for critical model improvement. More research can be initiated on this aspect.

6 RELATED WORK

The work studied in this paper relates to quality improvement for DL models and applications. We briefly discuss representative work in recent years, on data quality assurance, DL-related verification, testing and debugging, and adversarial attack.

Data quality assurance. One line of work focuses on assuring quality data for DL models, so as to improve their reliability in use.

Some work [4, 19–21] focused on data cleaning techniques in order to prepare qualified training data for instantiating models. Other work [13, 28, 30, 31] regarded training samples as in-distribution and refined given new samples by identifying out-of-distribution ones as outliers a statistic way. This line of work typically focuses on data characteristics, and seldom considers requirements from applications built on these data.

DL-related verification. This line of work attempts to formally verify DL models for their quality. Some work [17, 49] proposed to use symbolic techniques for abstracting the input space for a DL model, but could hardly scale to large and complex ones. Some [9, 15, 41] could work for relatively larger models, but supported only specific neural networks. The others [14, 48, 59, 67] targeted at security-critical or safety-critical DL-assisted applications, and verified them for safety properties. Our work complements this line, by validating inputs at runtime for better model performance.

Testing and debugging. This third line of work aims to generate diverse and realistic corner cases to expose possible flaws in DL models. For example, DeepXplore [47] converted the corner-case generation problem to a joint optimization one, and searched towards a model's decision boundaries. More work examined DL models by testing, e.g., by image transformation [57], input fuzzing [11], mutation testing [35], and metamorphic testing [66]. To measure the testing adequacy, various coverage criteria were proposed, including the neuron coverage [47, 57], SS coverage [54], neuron- and layer-level coverage [34], combination dense coverage [36], and surprise coverage [18]. Interestingly, it was also argued that such structural coverage criteria for DL models could be misleading [29]. Finally, DL models could be debugged for their flaws by analyzing biased data distribution [37]. Our work also complements this line, by runtime validation for tolerating unexpected problems after the concerned DL models have been tested and deployed.

Adversarial attack. Finally, proposing better DL models against adversarial attacks is getting hotter. On one hand, adversarial attacking techniques were proposed to generate adversarial samples with small artificial perturbations to fool DL models, e.g., L-BFGS [56], FGSM [10], JSMA [45], C&W [3], Uni. perturbation [39], and DeepFool [40]. On the other hand, defense techniques were also proposed to identify such fooling samples, e.g., adversarial sample detection [58], adversarial training [56], foveation-based defense [32], gradient regularization/masking [33, 42, 50], defensive distillation [44, 46], and GAN-based defense [27]. They together improve DL models in a recursive way. Our work can also be used for identifying adversarial attacking samples as one of its application.

7 CONCLUSION

In this paper, we proposed Dissector for effective and efficient validation of inputs to DL applications. Dissector distinguishes beyond-inputs from within-inputs by collecting prediction uniqueness evidence, and works in a model-aware and lightweight way. The experimental evaluation confirmed Dissector's unique effectiveness and clear gains over existing techniques. Dissector also exhibited encouraging usefulness in defensing against adversarial inputs and improving a DL model' actual accuracy in use.

Currently, Dissector has to be configured for its growth type and parameters, although they do not affect much ($< 5\%$). We are considering guiding the configuration by prediction feedbacks of its generated sub-models to make Dissector fully automated. Besides, Dissector needs to be tested in more application scenarios, for both more practical validation and more usefulness exploration.

ACKNOWLEDGMENTS

This work was supported in part by National Key R&D Program (Grant #2017YFB1001801) and National Natural Science Foundation (Grants #61932021, #61690204, and #61802170) of China. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

[1] Naveed Akhtar and Ajmal Mian. 2018. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access* 6 (2018), 14410–14430.

[2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).

[3] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP 2017)*. IEEE, 39–57.

[4] Xu Chu, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing. (2015).

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2009)*. Ieee, 248–255.

[6] Reuben Feinman, Ryan R Curtin, Saurabh Shintre, and Andrew B Gardner. 2017. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410* (2017).

[7] JoÃčo Gama, IndrÄŮ Å¡liobaitÄŮ, Albert Bifet, Mykola Pechenizkiy, and Hamid Bouchachia. 2014. A Survey on Concept Drift Adaptation. *ACM Computing Surveys (CSUR)* 46 (04 2014). https://doi.org/10.1145/2523813

[8] Huang Gao, Liu Zhuang, Laurens Van Der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2017).*

[9] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*. 315–323.

[10] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[11] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, 739–743.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016)*. 770–778.

[13] Dan Hendrycks and Kevin Gimpel. 2016. A Baseline for Detecting Misclassified and Out-of-Distribution Examples in Neural Networks. (10 2016).

[14] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety verification of deep neural networks. In *Proceedings of the 2017 International Conference on Computer Aided Verification (CAV 2017)*. Springer, 3–29.

[15] Kevin Jarrett, Koray Kavukcuoglu, Yann LeCun, et al. 2009. What is the best multi-stage architecture for object recognition?. In *Proceedings of the 2009 IEEE 12th International Conference on Computer Vision (ICCV 2009)*. IEEE, 2146–2153.

[16] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. 2016. Policy compression for aircraft collision avoidance systems. In *Proceedings of the 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC 2016)*. IEEE, 1–10.

[17] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proceedings of the 2017 International Conference on Computer Aided Verification (CAV 2017)*. Springer, 97–117.

[18] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41th ACM/IEEE International Conference on Software Engineering (ICSE 2019), forthcoming, arXiv preprint arXiv:1808.08444.*

[19] Sanjay Krishnan, Michael J. Franklin, Ken Goldberg, Jiannan Wang, and Eugene Wu. 2016. ActiveClean: An Interactive Data Cleaning Framework For Modern Machine Learning. In *Proceedings of the 2016 ACM SIGMOD InternationalConference on Management of Data (SIGMOD'16)*.

[20] Sanjay Krishnan, Jiannan Wang, Michael J Franklin, Ken Goldberg, Tim Kraska, Tova Milo, and Eugene Wu. 2015. SampleClean: Fast and Reliable Analytics on Dirty Data. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 38, 3 (2015), 59–75.

[21] Sanjay Krishnan and Eugene Wu. 2019. AlphaClean: Automatic Generation of Data Cleaning Pipelines. (04 2019).

[22] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.

[23] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. 2016. Adversarial machine learning at scale. *arXiv preprint arXiv:1611.01236* (2016).

[24] Jonathan Laserson. 2011. From Neural Networks to Deep Learning: zeroing in on the human brain. *XRDS Crossroads the ACM Magazine for Students* 18, 1 (2011), 29–34.

[25] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

[26] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[27] Hyeungill Lee, Sungyeob Han, and Jungwoo Lee. 2017. Generative adversarial trainer: Defense to adversarial perturbations with GAN. *arXiv preprint arXiv:1705.03387* (2017).

[28] Kimin Lee, Kibok Lee, Honglak Lee, and Jinwoo Shin. 2018. A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks. In *Advances in Neural Information Processing Systems 31 (NIPS 2018)*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 7167–7177.

[29] Zenan Li, Xiaoxing Ma, Chang Xu, and Chun Cao. 2019. Structural Coverage Criteria for Neural Networks Could Be Misleading. In *Proceedings of the 41th ACM/IEEE International Conference on Software Engineering (ICSE 2019 NIER), forthcoming*.

[30] Shiyu Liang, Yixuan Li, and R. Srikant. Principled detection of out-of-distribution examples in neural networks. (????).

[31] Shiyu Liang, Yixuan Li, and R. Srikant. 2018. Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks. (2018).

[32] Yan Luo, Xavier Boix, Gemma Roig, Tomaso Poggio, and Qi Zhao. 2015. Foveation-based mechanisms alleviate adversarial examples. *arXiv preprint arXiv:1511.06292* (2015).

[33] Chunchuan Lyu, Kaizhu Huang, and Hai-Ning Liang. 2015. A unified gradient regularization family for adversarial examples. In *Proceedings of the 2015 IEEE International Conference on Data Mining (ICDM 2015)*. IEEE, 301–309.

[34] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. DeepGauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, 120–131.

[35] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. DeepMutation: Mutation testing of deep learning systems. In *Proceedings of the 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE 2018)*. IEEE, 100–111.

[36] Lei Ma, Fuyuan Zhang, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. Combinatorial testing for deep learning systems. *arXiv preprint arXiv:1806.07723* (2018).

[37] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, 175–186.

[38] Richard Mankiewicz. 2005. The story of mathematics. *Princeton University Press Princeton Nj* 9 (2005).

[39] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. 2017. Universal adversarial perturbations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2017)*. 1765–1773.

[40] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. DeepFool: a simple and accurate method to fool deep neural networks. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016)*. 2574–2582.

[41] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML 2010)*. 807–814.

[42] Linh Nguyen, Sky Wang, and Arunesh Sinha. 2018. A Learning and Masking Approach to Secure Learning. In *Proceedings of the 9th International Conference on Decision and Game Theory for Security (GameSec 2018)*. Springer, 453–464.

[43] Michael A Nielsen. 2015. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA.

[44] Nicolas Papernot and Patrick McDaniel. 2017. Extending defensive distillation. *arXiv preprint arXiv:1705.05264* (2017).

[45] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS&P 2016)*. IEEE, 372–387.

[46] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. Distillation as a defense to adversarial perturbations against deep neural networks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP 2016)*. IEEE, 582–597.

[47] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017)*. ACM, 1–18.

[48] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Towards practical verification of machine learning: The case of computer vision systems. *arXiv preprint arXiv:1712.01785* (2017).

[49] Luca Pulina and Armando Tacchella. 2010. An abstraction-refinement approach to verification of artificial neural networks. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV 2010)*. Springer, 243–257.

[50] Andrew Slavin Ross and Finale Doshi-Velez. 2018. Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018)*.

[51] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[52] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484.

[53] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[54] Youcheng Sun, Xiaowei Huang, and Daniel Kroening. 2018. Testing Deep Neural Networks. *arXiv preprint arXiv:1803.04792* (2018).

[55] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. (09 2014).

[56] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[57] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*. ACM, 303–314.

[58] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. 2019. Adversarial Sample Detection for Deep Neural Network through Model Mutation Testing. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019), forthcoming, arXiv preprint arXiv:1812.05793*.

[59] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 2018)*. 1599–1614.

[60] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR 2017)*. 1492–1500.

[61] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. 2016. Achieving human parity in conversational speech recognition. *arXiv preprint arXiv:1610.05256* (2016).

[62] Yang You, Zhao Zhang, Cho-Jui Hsieh, and James Demmel. 2017. 100-epoch ImageNet Training with AlexNet in 24 Minutes. *CoRR* abs/1709.05011 (2017). arXiv:1709.05011 http://arxiv.org/abs/1709.05011

[63] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droidsec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review (CCR 2014)*, Vol. 44. ACM, 371–372.

[64] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide Residual Networks. (05 2016).

[65] Chen-Lin Zhang, Jian-Hao Luo, Xiu-Shen Wei, and Jianxin Wu. 2018. *In Defense of Fully Connected Layers in Visual Representation Transfer*. 807–817. https://doi.org/10.1007/978-3-319-77383-4_79

[66] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, 132–142.

[67] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. 2019. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, 686–701.